თემა 3. # სწრაფი დახარისხება

განხილული საკითხები:

- სწრაფი დახარისხების ალგორითმი
- სწრაფი დახარისხების ალგორითმის სისწრაფე >>>
- სწრაფი დახარისხების ალგორითმის საუკეთესო განხორციელების გზები >>>

სწრაფი დახარისხების ალგორითმი

ეს არის რეკურსიული ალგორითმი, რომლის იდეა დაფუძნებულია შემდეგ დაკვირვებაზე: თუ ავიღებთ ზრდადობით დახარისხებული n ელემენტიანი $a=\{a_1,a_2,\ldots,a_n\}$ მიმდევრობის რაიმე $a_p,a_{p+1},\ldots,a_r$ ფრაგმენტს და q ნომერს ($p \le q < r$), ვნახავთ რომ $a_p,\ldots,a_q$ მიმდევრობის ნებისმიერი რიცხვი ნაკლებია ან ტოლი $a_{q+1},\ldots,a_r$ მიმდევრობის ნებისმიერ რიცხვზე.

ვთქვათ, არ ვიცით $a=\{a_1,a_2,\ldots,a_n\}$ მიმდევრობა არის თუ არა დახარისხებული. მაშინ მისი ნებისმიერი $a_p,a_{p+1},\ldots,a_r$ ფრაგმენტი (რომელიც, თავის მხრივ, ისევ მიმდევრობას წარმოადგენს) **სწრაფი დახარისხების ალგორითმის** მიხედვით დახარისხდება შემდეგნაირად:

- $a_p,a_{p+1},\ldots,a_r$ მიმდევრობის ელემენტები ისე გადანაწილდება, რომ რომ $a_p,\ldots,a_q$ სიმრავლის ყოველი ელემენტი ნაკლებია $a_{q+1},\ldots,a_r$ მიმდევრობის ნებისმიერ ელემენტზე. ამ ბიჯს ვუწოდოთ გადანაწილება და მისთვის ვიყენებთ STL ბიბლიოთეკის partition ალგორითმს. ბიჯის დასრულება ნიშნავს მარჯვენა მიმდევრობის პირველი ელემენტის პოზიციის დამახსოვრებას. მაგრამ, თუ a_p საწყისი ფრაგმენტის მინიმალური ელემენტია, მაშინ მარცხენა ფრაგმენტი ცარიელია. ამ დროს საწყისი სიმრავლეს ვყოფთ ასე: ერთელემენტიანი მიმდევრობა a_p და ყველა დანარჩენები ერთად.
- სწრაფი დახარისხების ალგორითმის რეკურსიული გამოძახება ხდება ორივე მიმდევრობისთვის ცალ-ცალკე. თუ მიმდევრობაში მხოლოდ ერთი ელემენტია, რეკურსიული გამოძახებები წყდება.

ცხადია, ალგორითმი ზრდადობით დაახარისხებს $a=\{a_1,a_2,\ldots,a_n\}$ მიმდევრობას. ეს შეგვიძლია დავამტკიცოთ მათემატიკური ინდუქციით. როცა ერთი ელემენტია, ალგორითმი არაფერს ცვლის. დავუშვათ, ალგორითმი ზრდადობით ახარისხებს ნებისმიერ n ელემენტიან მიმდევრობას. ავიღოთ რაიმე (n+1) ელემენტიანი მიმდევრობა. პირველივე გადანაწილება ორ ნაწილად დაყოფს მას, უფრო დიდში ვერ იქნება n ელემენტზე მეტი. ამიტომ, იგივე ალგორითმის რეკურსიული გაშვება ზრდადობით დაახარისხებს ორივე ფრაგმენტს. მარცხენას მაქსიმუმი მოექცევა ბოლოში, მარჯვენას მინიმუმი თავში, ანუ მეზობლად მოხვდებიან. მაგრამ, რადგან მარჯვენას ნებისმიერი ელემენტი მეტია ან ტოლი მარცხენის ნებისმიერ ელემენტზე, ამიტომ ეს ორი ელემენტი კერძოდ და მთელი საწყისი მიმდევრობაც ზრდადობითაა დახარისხებული. n-ის ნებისმიერობის ძალით ალგორითმის კორექტულობა დამტკიცებულია.

$a_p,\ldots,a_q$ ფრაგმენტის დასაწყისი აღვნიშნოთ *first*-თი (და გავიაზროთ იტერარტორების ტერმინებში), ხოლო მისი დასასრული *last*-ით. ამ ფრაგმენტის სწრაფი დახარისხების ალგორითმი აღვნიშნოთ QUICKSORT(*first*,*last*)-ით, ხოლო იგივე ფრაგმენტის დაყოფის ალგორითმი (დეტალებზე კოდის განხილვისას ვისაუბრებთ) PARTITION(*first*,*last*)-ით, მაშინ სწრაფი დახარისხების ალგორითმის ფსევდოკოდს აქვს სახე:

```
QUICKSORT(first,last)
1     if ( distance(first,last)>1 )    //ფრაგმენტში ერთზე მეტი ელემენტია
2          q = PARTITION(first,last);    if(first == q)  q = next(p);
3          QUICKSORT (last, q);
4          QUICKSORT (q , last);
```

ჩანაწერი `QUICKSORT (first,q)` ნიშნავს, რომ `q` პოზიციაზე მდგომი ელემენტი არ განიხილება, რადგან ნახევრადღია შუალედებს განვიხილავთ ხოლმე იტერატორების შემთხვევაში. რაიმე `a` კონტეინერის დახარისხებისთვის საკმარისია გამოვიძახოთ `QUICKSORT(a.begin(),a.end())`.

სწრაფი დახარისხების ალგორითმის ძირითად ბიჯს წარმოადგენს ალგორითმი (ფუნქცია) `PARTITION`. ძალიან მოსახერხებელია მისი რეალიზაცია STL ბიბლიოთეკის

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p)
{
1       first = std::find_if_not(first, last, p);
2       if (first == last) return first;

3       for (ForwardIt i = std::next(first); i != last; ++i) {
4           if (p(*i)) {
5               std::iter_swap(i, first);
6               ++first;
            }
        }
7       return first;
}
```

ალგორითმის საფუძველზე. ამ მიზნით, ყოველი `[first,last)` დიაპაზონისათვის პრედიკატი უნდა გავაკეთოთ ასე: თუ ელემენტი ნაკლებია `*first` მნიშვნელობაზე, მაშინ იგი აბრუნებს ჭეშმარიტს, წინააღმდეგ შემთხვევაში მცდარს. ეს ალგორითმი განიხილება პრაქტიკულ მეცადინეობაზე, ამიტომ მხოლოდ მის ხანგრძლივობაზე (სისწრაფეზე) შევჩერდებით. ვთქვათ დიაპაზონში არის `n` ცალი ელემენტი. მაშინ, სტრიქონები 1 და 3 ერთდროულად არიან $\Theta(n)$ რიგის. რადგან ყოველი `iter_swap` იწვევს იტერატორის `first` გაზრდას, ამიტომ `iter_swap`-ებისა და `++first`-ების მთლიანი რაოდენობა ასევე არის $\Theta(n)$ რიგის. საბოლოოდ, ალგორითმის ხანგრძლივობაც ესაა.

<<< სწრაფი დახარისხების ალგორითმის სისწრაფე

აღვნიშნოთ $T(n)$-ით `n` ელემენტიანი მასივისთვის სწრაფი დახარისხების ალგორითმის სისწრაფე. ცხადია, ერთელემენტიანი მასივისთვის $T(1)=c_1=\Theta(1)$. ზოგად შემთხვევაში, თუ მასივი იყოვა i და $n-i$ ელემენტებიან ნაწილებად,

$$T(n) = \Theta(n) + T(i) + T(n-i),$$

სადაც $\Theta(n)$ არის n რიგის რაიმე ფუნქცია, რომელიც აღწერს `n` ელემენტიანი მასივისთვის `PARTITION` ალგორითმის მუშაობის სისწრაფეს.

არსაიდან არ გამომდინარეობს, რომ ყოველ ბიჯზე მასივის დაყოფა მაინცდამაინც ერთნაირი პროპორციით მოხდება.

განვიხილოთ **უარესი შემთხვევა**, როდესაც გაყოფა მაქსიმალურად არათანაბარია, ანუ ერთ-ერთ ჯგუფში მხოლოდ ერთი ელემენტი ხვდება. ასეთი შემთხვევა გვაქვს, როცა ზრდადობით ან კლებადობით დახარისხებული დიაპაზონის ხელახლა დახარისხებას ვცდილობთ ზრდადობით. ამ დროს,

$$T(n) = T(n-1) + \Theta(n),$$

რაც $T(1)=\Theta(1)$-ის გათვალისწინებით გვაძლევს:

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n}\Theta(k) = \Theta\left(\sum_{k=1}^{n}k\right) = \Theta(n^2).$$

იმისათვის რომ უფრო არაფორმალურად შევაფასოთ ეს ჯამი, ყოველი $\Theta(k)$-ს ნაცვლად უნდა ავიღოთ ამ კლასის რომელიმე კონკრეტული ფუნქცია, მაგალითად პირდაპირ k.
ახლა განვიხილოთ **საუკეთესო შემთხვევა**, როცა ყოველ ბიჯზე დიაპაზონი თანაბრად იყოფა:

$$T(n) = 2\cdot T(n/2)+\Theta(n).$$

$\Theta(n)$-ის ნაცვლად ავიღოთ პირდაპირ n და შევაფასოთ სისწრაფე მიმდევრობითი ჩასმებით:

$$T(n) = 2\cdot T(n/2)+n = 2\cdot\left(2\cdot T(n/4)+n/2\right)+n = 4\cdot T(n/4)+2\cdot n =$$

(და ა.შ., თუ n-ს გავანახევრებთ $\lfloor \log n \rfloor$-ჯერ, იგი გახდება ერთის ტოლი, ამიტომ)

$$= 2^{\lfloor \log n \rfloor}\, T(1)+\lfloor \log n \rfloor \cdot n,$$

ანუ

$$T(n) = = \Theta(n)\Theta(1)+\Theta(\log n)\Theta(n) = \Theta(n\log n).$$

საშუალო სისწრაფის შეფასება საშუალოზე მაღალი სირთულის ამოცანაა და მისი დამტკიცება სცილდება ჩვენი კურსის მიზანს. საკმარისია შევნიშნოთ, რომ საშუალო და საუკეთესო შემთხვევებში სისწრაფე ერთი და იგივე რიგისაა.

<<< სწრაფი დახარისხების ალგორითმის საუკეთესო განხორციელების გზები

განხილულ ალგორითმში, ყოველ რეკურსიულ ბიჯზე გამყოფ ელემენტად ვირჩევთ დიაპაზონის ელემენტებიდან ყველაზე მარცხენას და შემდეგ ვახდენდით ელემენტების გადაჯგუფებას ორ ნაწილად ისე, რომ ერთში ხვდებოდა გამყოფ ელემენტზე ნაკლები ან ტოლი ყველა რიცხვი, ხოლო მეორეში - მეტი ან ტოლი.

ამ ალგორითმის მრავალი ვარიაცია არსებობს, რადგან გამყოფად ჩვენ შეგვიძლია ავირჩიოთ არა აუცილებლად მარცხენა ელემენტი, არამედ რომელიმე გაჩერების პირობებიც შეიძლება განსხვავებულად იქნას შერჩეული. მაგალითად, გამყოფ ელემენტად ავირჩიოთ დიაპაზონის შუა ელემენტი.

სწრაფი დახარისხების ალგორითმის საუკეთესო რეალიზაციები ზოგად შემთხვევაში ყველაზე სწრაფ დახარისხების ალგორითმებს წარმოადგენენ. მაგრამ საუკეთესო რეალიზაცია რთული შესაქმნელია. პირველ რიგში, იცვლება პირობა

```
if ( distance(p,r)>1 );
```

თუ ფრაგმენტში 20 ან ნაკლები ელემენტია, მაშინ იყენებენ კვადრატული სისწრაფის მქონე ალგორითმს, რომელიც ელემენტების მცირე რაოდენობისთვის (20 და ნაკლები) ძალიან სწრაფია.

თუ ფრაგმენტში 20-ზე მეტი ელემენტია, მაშინ აქედან ნებისმიერ სამს ამოარჩევენ და პოულობენ მათს საშუალო არითმეტიკულს, ვთქვათ x. ეს სიდიდე წარმოადგენს გამყოფ ელემენტს: გადანაწილების ალგორითმში ყოველი [first,last) დიაპაზონისათვის პრედიკატი კეთდება ასე: თუ ელემენტი ნაკლებია x მნიშვნელობაზე, მაშინ იგი აბრუნებს ჭეშმარიტს, წინააღმდეგ შემთხვევაში - მცდარს.

ასეთი გაუმჯობესების ფასი იმაში მდგომარეობს, რომ ფრაგმენტიდან რაიმე ნებისმიერი ელემენტის ამოღებას სჭირდება სწრაფი, ანუ ნებისმიერი წვდომა, ანუ იტერატორები უნდა იყოს სწრაფი წვდომის მაინც.